下载APP

20 | Mock：如何屏蔽第三方接口的影响？

2021-12-02 高楼

《全链路压测实战30讲》 课程介绍 >

讲述：高楼

时长 15:52 大小 14.54M

你好，我是高楼。这节课，我们聊一聊全链路压测中 Mock 技术的落地。

曾经也有不少人问我，在压测过程中，第三方接口响应时间慢怎么办呢？其实，响应时间慢，容量达不到要求，这是在依赖第三方系统的全链路压测过程中必然会面对的问题。这个时候，性能测试还要做，开发也不会为了性能测试修改业务代码，第三方我们也驱动不了。为了解决这个问题，我们就不得不用上 Mock 技术了。

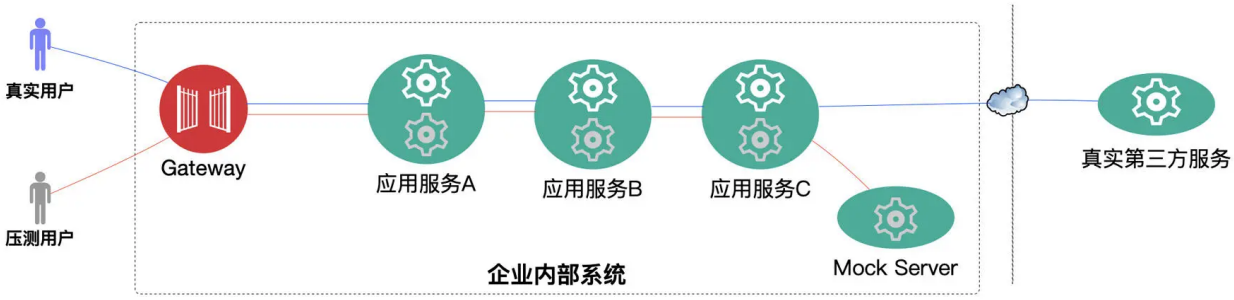
什么是 Mock 呢？简单来说就是指使用各种技术手段模拟出各种需要的资源以供测试使用。Mock 技术目前按应用场景主要分两大类：

Mock 一个对象，构造返回预期的数据，主要适用于单元测试；

Mock 一个 Server ，构造返回预期的服务，主要适用于接口和性能测试。

而对于我们全链路压测来说，最主要使用的技术还是 Mock Server。

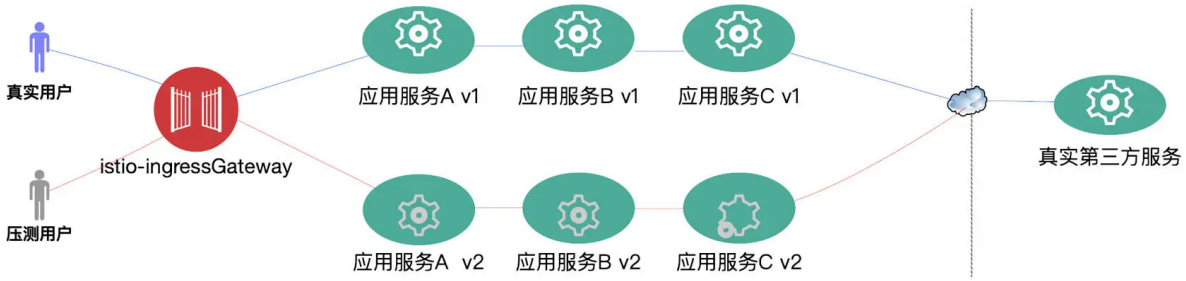
Mock Server 的逻辑是非常直观的：



通过这张图我们可以看到，真实用户和压测用户是走同样的应用服务节点的。但不同的是，真实用户最终会走到真实的第三方服务，而压测用户会走到 Mock Server 上去。

请注意，在全链路压测的过程中，我们要 Mock 的是压测流量。而在整个逻辑中，我们是直接在应用服务中做标记透传、识别、流量隔离的，所以压测流量和正式流量是用**同样的应用服务**。

说到这里，我还想提一下 Service Mesh。现在网上我们经常看到的通过 Service Mesh 实现的全链路逻辑，是在入口的 POD 上添加 tag，并通过网络路由实现对不同版本的转发的。这个逻辑其实也是实现了全链路方式的。逻辑图如下：



但是，这种方式有两个点要说明：

1. Service Mesh 是用不同的应用服务容器来实现的。也就是说，压测流量和正常流量走的已经不是同一个 POD 中的应用服务了。而我们专栏描述的内容，是会走同样的 POD 中的应用服务的。
2. Service Mesh 的方式无法实现对数据库、缓存、队列等的隔离。

所以从这两点来看，Service Mesh 做灰度发布是完全没问题的，但是要想实现全链路压测的初衷它是做不到的。

也正是因此，我们才需要用 Mock 的技术手段来做第三方的服务隔离。

虽然 Mock 技术并不复杂，但它却是全链路压测过程中不可或缺的一个知识点，所以我们还是把它拆开讲一讲。

市面上有很多种前后端 Mock Server 框架，我们只要根据项目需要搭建最简单实用的 Mock Server 框架解决问题就可以了。如果实在没有框架可以借鉴，我们还可以自己写一个 Mock Server，**反正 Mock 的目的就是把不可控变成可控**。

通常，我们把 Mock Server 分为**前端 Mock Serve**r 和**后端 Mock Server**。

不过，因为前端 Mock Server 主要是为了规避在开发过程中前端等待后端的情况。但是在全链路的逻辑中，使用 Mock Server 主要是为了规避调用第三方时的影响，所以在全链路压测项目中，我们只要实现后端 Mock Server 就可以了。

为了逻辑描述的完整性，这里我还是会给出一个前端 Mock Server 技术的简单示例，你可以稍做了解。

下面，我们就来看看前端 Mock Server 和后端 Mock Server 各有什么技术吧。

前端 Mock Server

当前端开发有依赖后端接口的数据时，我们可以在前端自己搭建 nodejs 服务，构造返回预期的服务。那么常见的前端 Mock Server 有哪些呢？

这里我给你提供两个常见的 Mock Server： server-mock 和 json-server 。

我们以 server-mock 为例，介绍一下它怎么使用。

第一步，安装完 nodejs 环境之后，执行 npm 快速安装 server-mock，命令如下：

复制代码

```
npm install -g server-Mock
```

第二步，在当前目录新建一个文件夹（名字随便，这里我用了 7d）。

再次输入命令 Mock init，输入完毕后，目录下自然会创建两个文件：index.html 与 router.js。

你可以打开这两个文件查看一下代码。这是一个很简单的 demo，index.html 就是一个表单与发送请求的 js 代码；router.js 内是 Mock 规则，当收到某个特定字段后，Mock Server 应该返回相应的响应信息。

第三步，简单修改下 index.html 文件：

```
<body>
  <div class="query-area">
    <input type="text" name="username" value="7dGroup" placeholder="7dGroup, 接口, anyone">
    <button>课程</button>
  </div>
  <div class="detail-area">
    <ul>
```

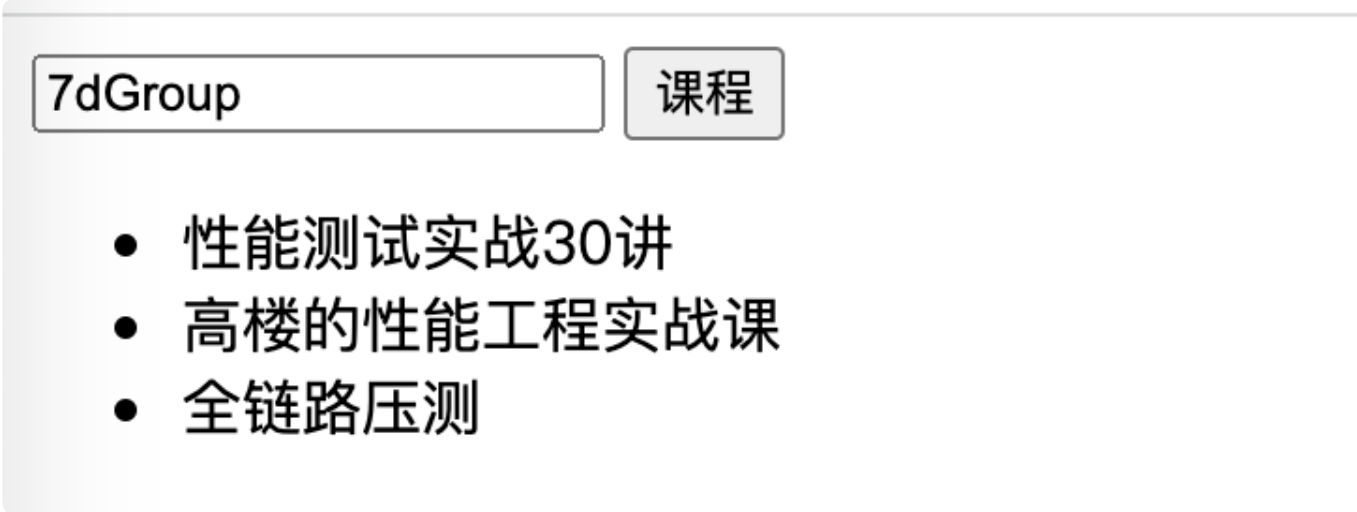
第四步，修改 route.js 文件，对调用的接口和参数设计对应的响应内容：

```
/**
 * 当 http://localhost:8080/getCourse 的GET请求到来时被下面匹配到进行处理
 * 通过req.query获取请求的参数对象
 * 通过 req.send发送响应
 */
router.get('/getCourse', function(req, res) {
  var username = req.query.username // 通过 req.query获取请求参数
  var course

  //根据请求参数mock数据
  switch (username){
    case '7dGroup':
    course = ['性能测试实战30讲', '高楼的性能工程实战课','全链路压测']
      break
    case '接口':
    course = ['java接口测试', 'python接口测','其他接口测试']
      break
    default:
    course = ['没有朋友']
  }
  res.send(course)
})

```

第五步，在命令行输入 Mock start 启动 server-mock， 终端上就会有下面的提示：

```
Success: server start success， open the link http://localhost:8080 in browser
```

我们只要根据提示在浏览器中打开地址就能看到效果了。



另外，你还可以直接在浏览器输入地址，响应结果就是 JSON 格式的自定义内容，如下所示：

有了接口响应数据，接下来就可以做前端开发了。

后端 Mock Server

好了，刚才，我们简单了解了一下前端 Mock Server ，并演示了 server-mock 的使用方法。接下来，就要进入我们这节课的重点也就是后端 Mock Server 了。

这类 Mock Server 有很多，下面我们重点介绍一款常用的：Moco。

简单介绍一下 Moco：

> Moco 本身支持 API 和独立运行两种方式。通过 API ，开发人员可以在 Junit、TestNg 等测试框架里使用 Moco，这样极大地降低了接口测试的复杂度。
> Moco 可以根据一些配置，启动一个真正的 HTTP 服务（监听本地指定端口）。当发起的请求满足一个条件时，就会收到一个 response 。Moco 底层并没有依赖于像 Servlet 这样的重型框架，而是基于 Netty 的网络应用框架编写的，这样就绕过了复杂的应用服务器，所以它的速度是极快的。

了解了 Moco 的基本特性，接下来我们就看下怎么实现。

首先第一步，下载 Moco 的 jar（https://github.com/dreamhead/moco ），启动 Moco HTTP Server。

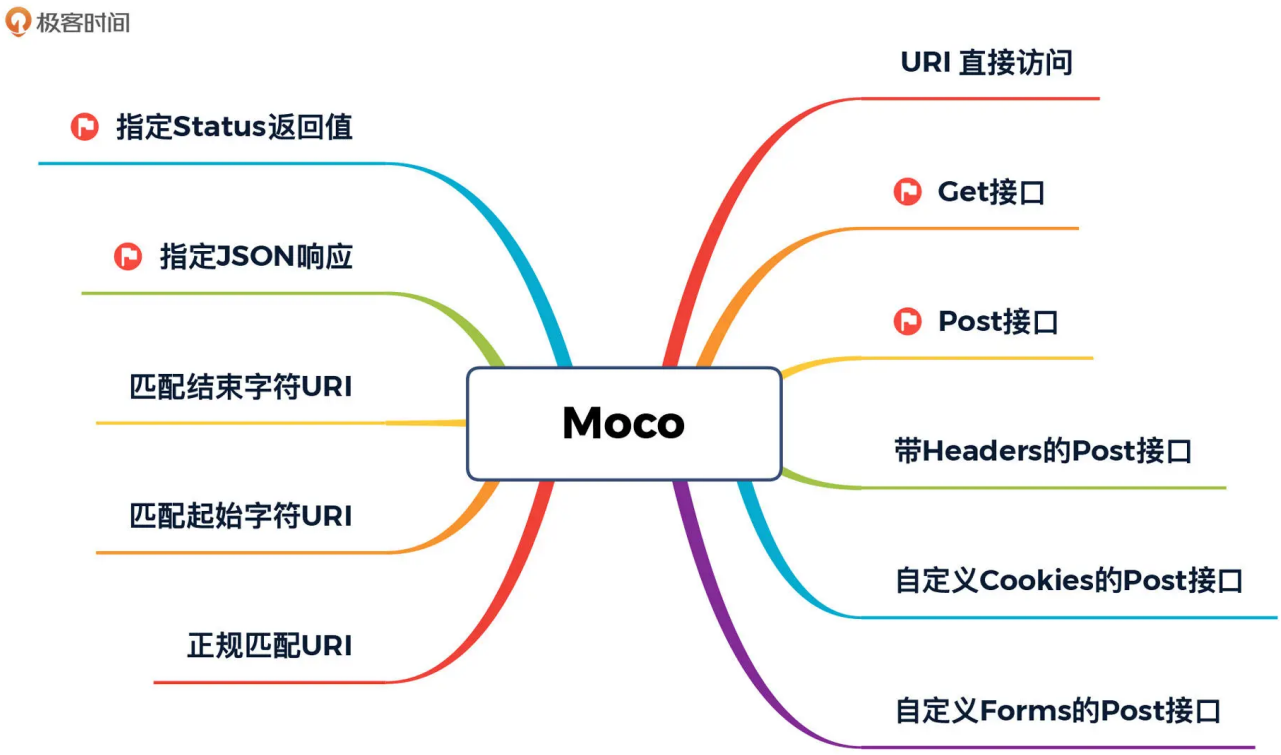
复制代码

```
1 java -jar <moco-runner-path> http -p <port> -c <configfile-path>
```

这个启动命令里，这几个参数的含义你可以了解一下：**moco-runner-path** 指 jar 包路径；**port** 指 HTTP 服务监听端口；**configfile-path** 指配置文件路径。

第二步，在本地启动一个 HTTP 服务器，其中监听端口是 12306，配置文件是 JSON 文件。只需要本机发起一个 request 就可以了，如：http://localhost:12306 。

根据不同的请求类型，我们要设计相应的返回值。我给你画了一个简单的思维导图，方便你理清不同类型的接口。



针对这张图中几个常见的接口（比如：Get 接口、Post 接口、带 Headers 的 Post 接口、自定义 Cookies 的 Post 接口、指定 JSON 响应、指定 Status 返回值），我们做一下具体的演示。

Get 接口

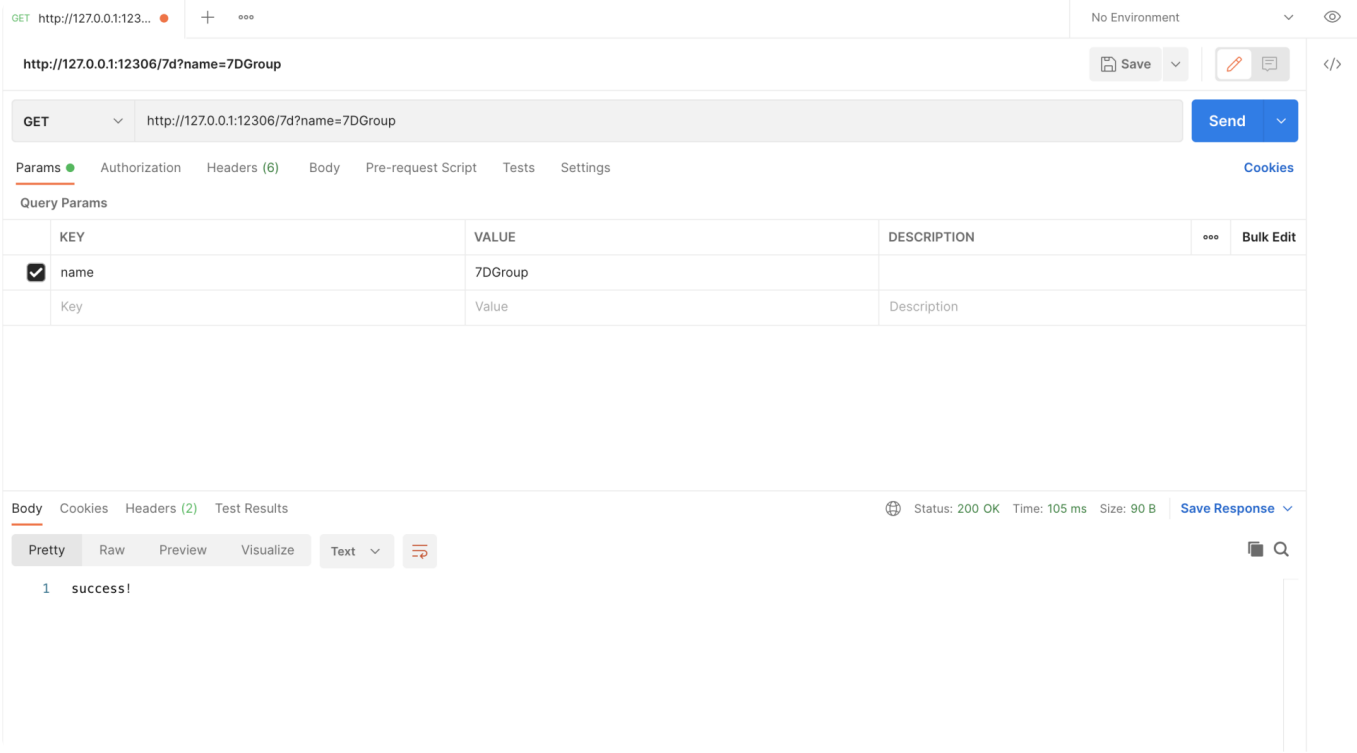
返回值 JSON 内容定义为：

复制代码

```
[
  {
    "description":"这是一个请求queries",
```

```
    "request":{
      "uri":"/7d",
      "queries":{
        "name":"7DGroup"
      }
    },
    "response":{
      "text":"success！"
    }
  }
]
```

通过 Postman 验证服务，测试 Get 请求：



Moco 服务日志为：

复制代码

```
09 十一月 2021 11:21:04 [nioEventLoopGroup-3-2] INFO  Request received:
GET /7d HTTP/1.1
Host: 127.0.0.1:12306
User-Agent: PostmanRuntime/7.4.0
Accept: */*
Accept-Encoding: gzip, deflate
Cache-Control: no-cache
Postman-Token: 2d36e386-e022-4478-8acd-258eff4ff684
X-Lantern-Version: 5.1.0
Content-Length: 0
```

```
09 十一月 2021 11:21:04 [nioEventLoopGroup-3-2] INFO  Response return:
HTTP/1.1 200
Content-Length: 10
Content-Type: text/plain; charset=utf-8
success!
```

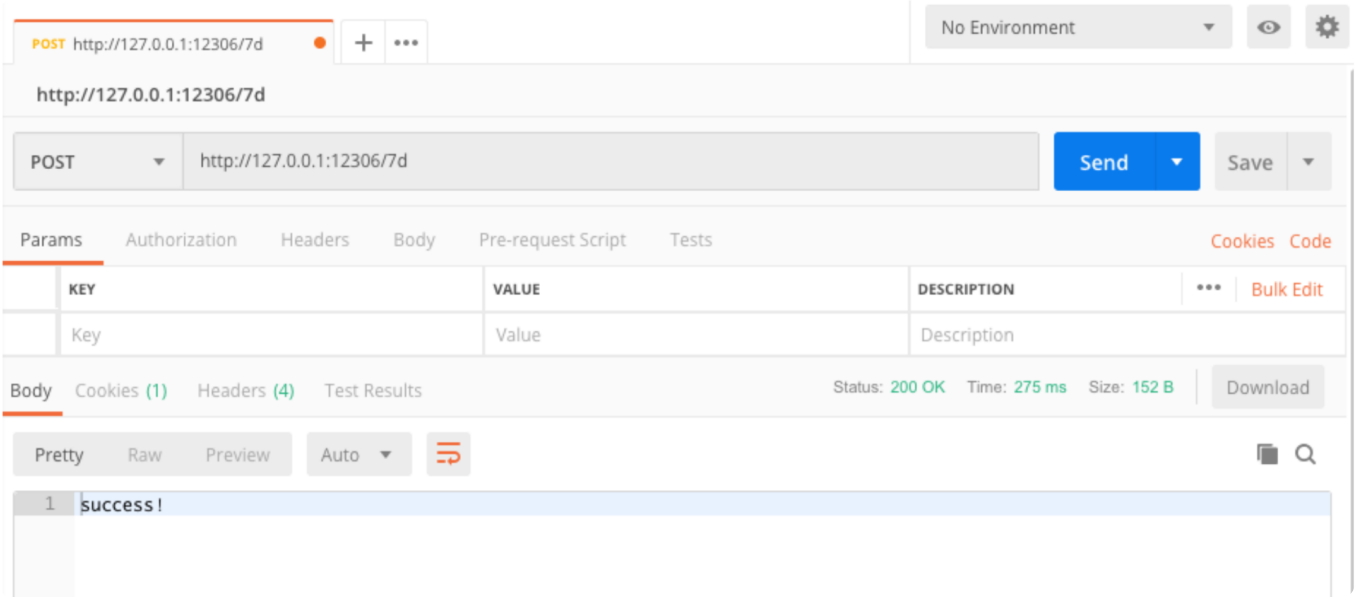
Post 接口

返回值 JSON 内容定义如下：

复制代码

```
[
  {
    "description":"这是一个post请求",
    "request":{
      "uri":"/7d",
      "method":"post"
    },
    "response":{
      "text":"success!"
    }
  }
]
```

通过 Postman 验证服务，测试 Post 请求：



Moco 服务日志为：

复制代码

```
09 十一月 2021 11:29:30 [nioEventLoopGroup-3-2] INFO  Request received:
POST /7d HTTP/1.1
Host: 127.0.0.1:12306
User-Agent: PostmanRuntime/7.4.0
Content-Length: 0
Accept: */*
Accept-Encoding: gzip, deflate
Cache-Control: no-cache
Postman-Token: 73f38af1-4efb-473a-b9d2-de0392c65bbe
X-Lantern-Version: 5.1.0
09 十一月 2021 11:29:30 [nioEventLoopGroup-3-2] INFO  Response return:
HTTP/1.1 200
Content-Length: 10
Content-Type: text/plain; charset=utf-8
success！
```

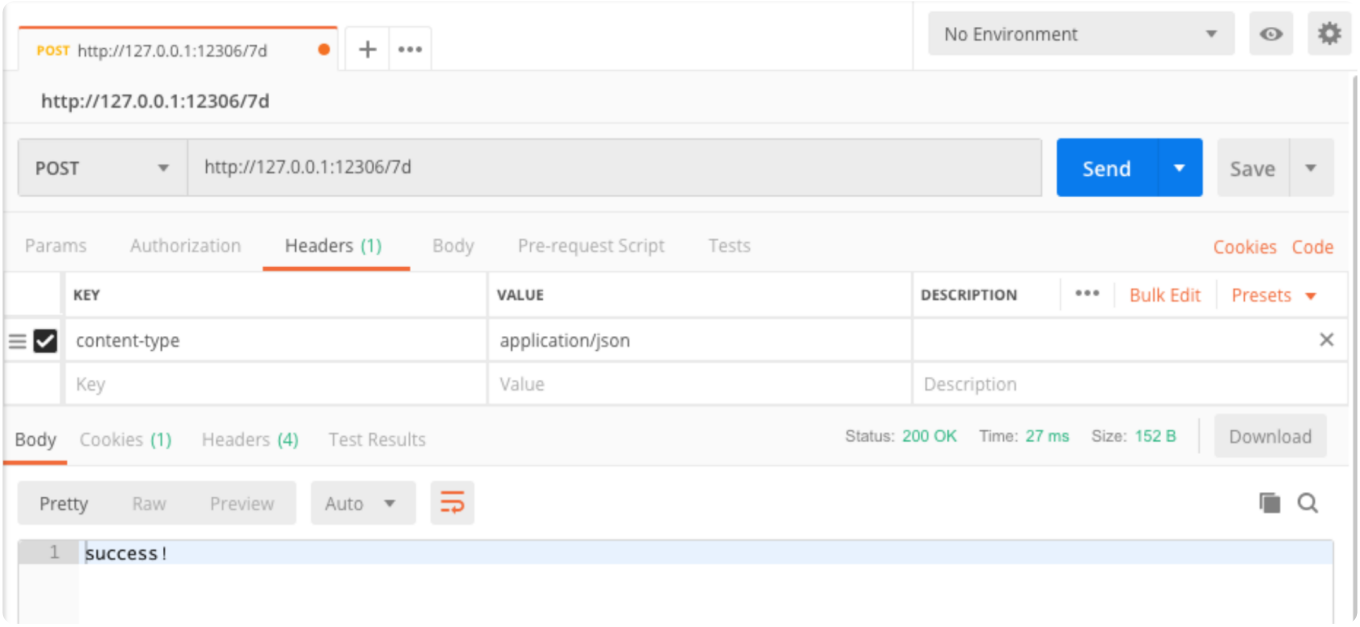
带 Headers 的 Post 接口

返回值 JSON 内容定义如下：

复制代码

```
[
  {
    "description":"这是一个带headers的post请求",
    "request":{
      "uri":"/7d",
      "method":"post",
      "headers":{
        "content-type":"application/json"
      }
    },
    "response":{
      "text":"success！"
      }
    }
]
```

通过 Postman 验证服务，测试带 **Headers** 的 Post 请求：



Moco 服务日志为：

复制代码

```
09 十一月 2021 11:34:43 [nioEventLoopGroup-3-2] INFO  Request received:
POST /7d HTTP/1.1
Host: 127.0.0.1:12306
User-Agent: PostmanRuntime/7.4.0
Content-Length: 0
Accept: */*
Accept-Encoding: gzip, deflate
Cache-Control: no-cache
Content-Type: application/json
Postman-Token: 0a82d74b-303f-42a3-9da0-32fd6c604166
X-Lantern-Version: 5.1.0
09 十一月 2021 11:34:43 [nioEventLoopGroup-3-2] INFO  Response return:
HTTP/1.1 200
Content-Length: 10
Content-Type: text/plain; charset=utf-8
success!
```

自定义 Cookies 的 Post 接口

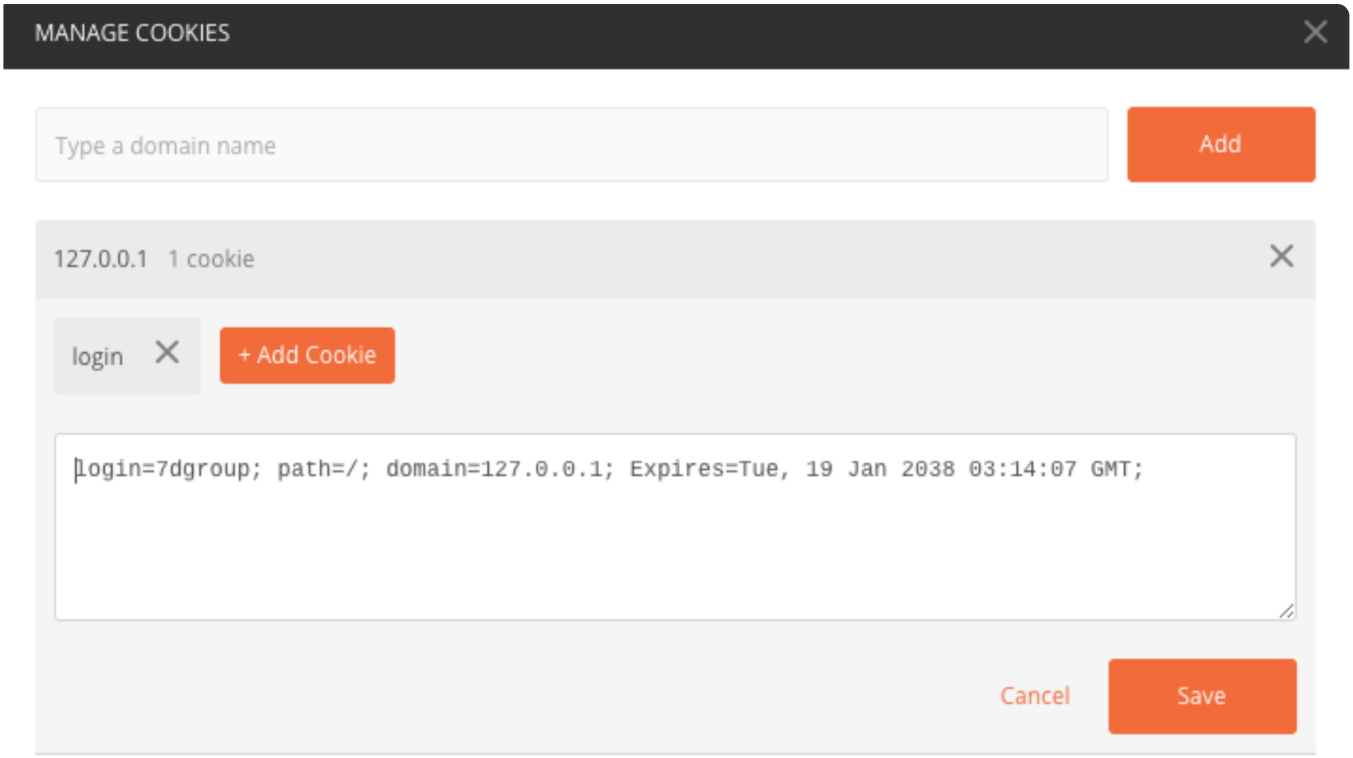
返回值 JSON 内容定义如下：

复制代码

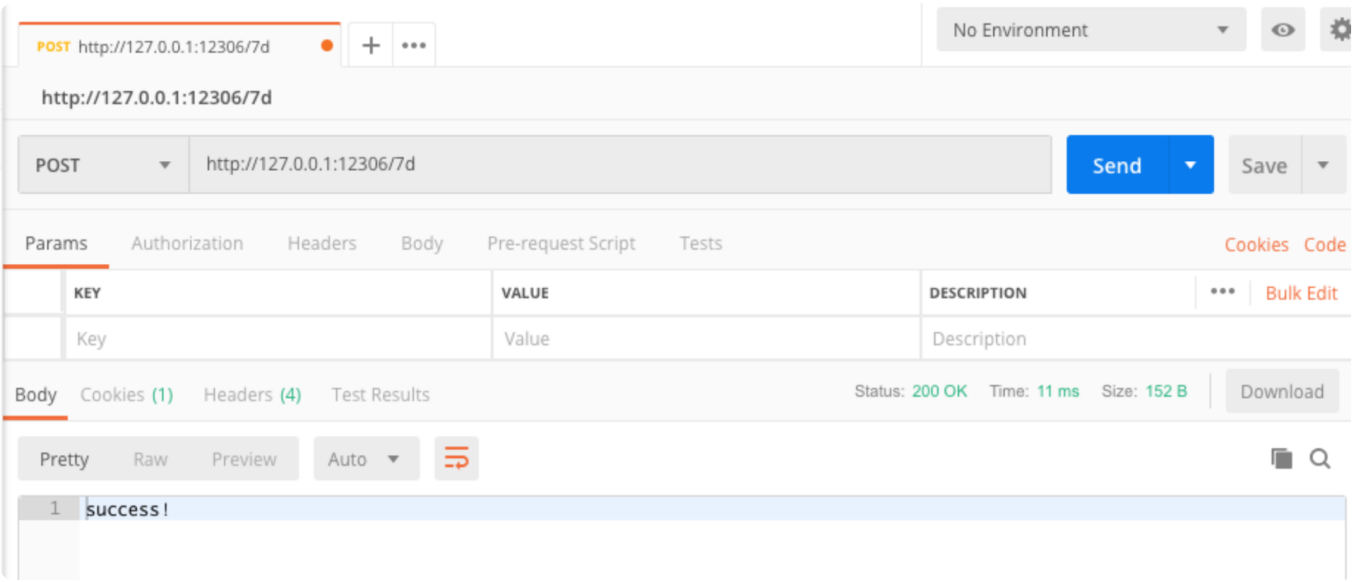
```
[
  {
    "description":"这是一个带cookies的post请求",
    "request":{
```

```
      "uri":"/7d",
      "method":"post",
      "cookies":{
        "login":"7dgroup"
      }
    },
    "response":{
      "text":"success！"
      }
    }
]
```

通过 Postman 验证服务，发送带自定义 Cookie 的 Post 请求，Post 请求中的 Cookie 配置如下：



Postman 接口配置如下：



Moco 服务日志如下：

复制代码

```
09 十一月 2021 12:26:46 [nioEventLoopGroup-3-3] INFO  Request received:

POST /7d HTTP/1.1
Host: 127.0.0.1:12306
User-Agent: PostmanRuntime/7.4.0
Content-Length: 0
Accept: */*
Accept-Encoding: gzip, deflate
Cache-Control: no-cache
Cookie: login=7dgroup
Postman-Token: 36a12412-6eb1-44a4-a2d8-ea222eba8968
X-Lantern-Version: 5.1.0

09 十一月 2021 12:26:46 [nioEventLoopGroup-3-3] INFO  Response return:

HTTP/1.1 200
Content-Length: 10
Content-Type: text/plain; charset=utf-8

success!
```

指定 JSON 响应

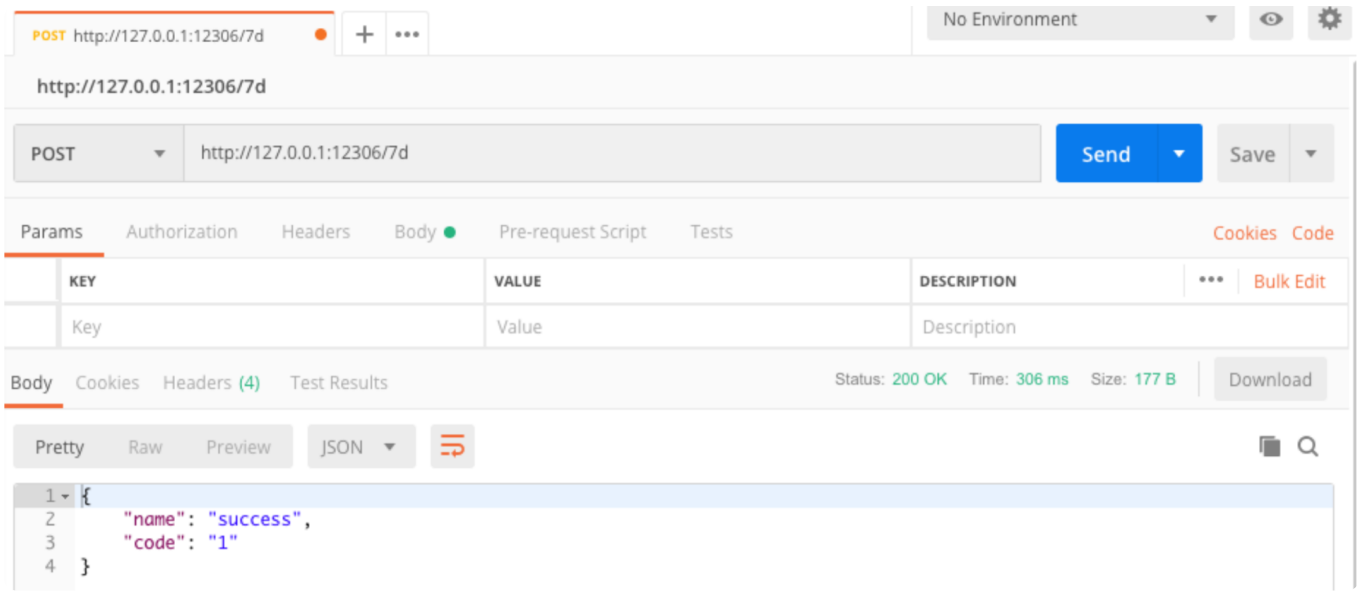
返回值 JSON 内容定义如下：

复制代码

```
[
```

```
  {
    "description":"这是一个指定Json响应的post请求",
    "request":{
      "uri":"/7d",
      "method":"post"
    },
    "response":{
      "json":{
        "name":"success",
        "code":"1"
      }
    }
  }
]
```

通过 Postman 验证服务，测试 Post 请求：



Moco 服务日志如下：

复制代码

```
09 十一月 2021 13:25:19 [nioEventLoopGroup-3-2] INFO  Request received:

POST /7d HTTP/1.1
Host: 127.0.0.1:12306
User-Agent: PostmanRuntime/7.4.0
Content-Length: 0
Accept: */*
Accept-Encoding: gzip, deflate
Cache-Control: no-cache
Content-Type: multipart/form-data; boundary=--------------------------70334172
Postman-Token: e5686919-85b9-44d0-8a73-61bf804b6377
X-Lantern-Version: 5.1.0
```

```
09 十一月 2021 13:25:19 [nioEventLoopGroup-3-2] INFO  Response return:

HTTP/1.1 200
Content-Length: 29
Content-Type: application/json; charset=utf-8

{"name":"success","code":"1"}
```

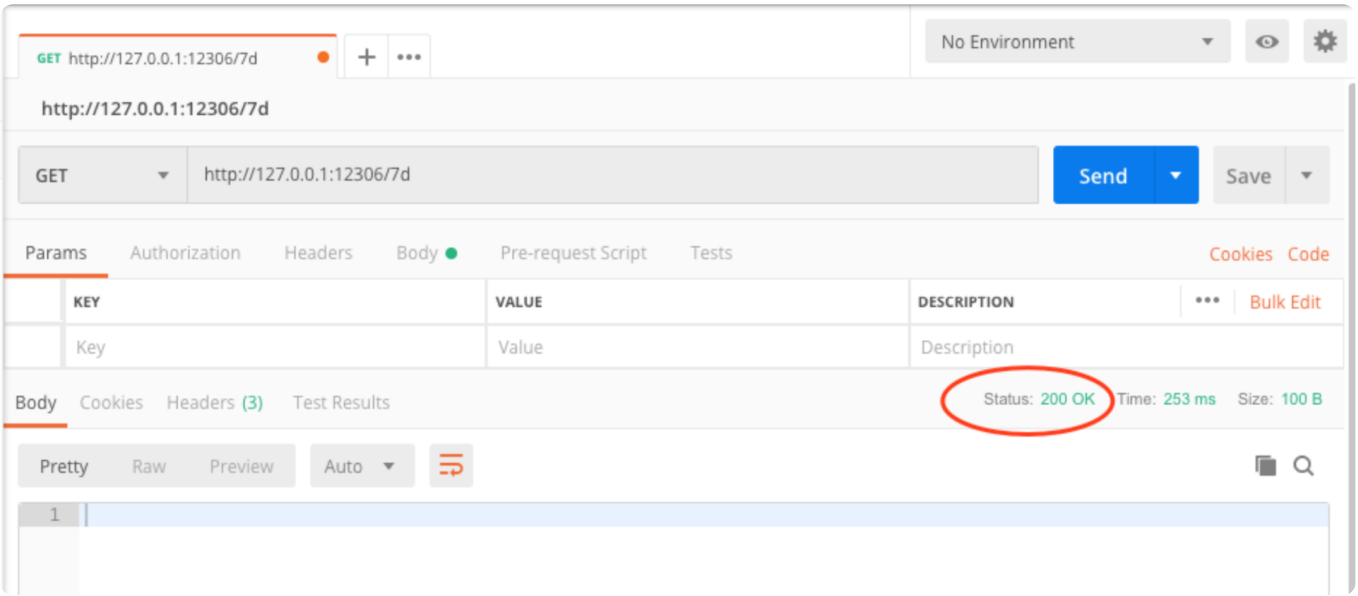
指定 Status 返回值

返回值 JSON 内容定义如下：

复制代码

```
[
  {
    "description":"这是指定响应status的get请求",
    "request":{
      "uri":"/7d",
      "method":"get"
    },
    "response":{
      "status":200
    }
  }
]
```

通过 Postman 验证服务，测试 Get 请求：



Moco 服务日志如下：

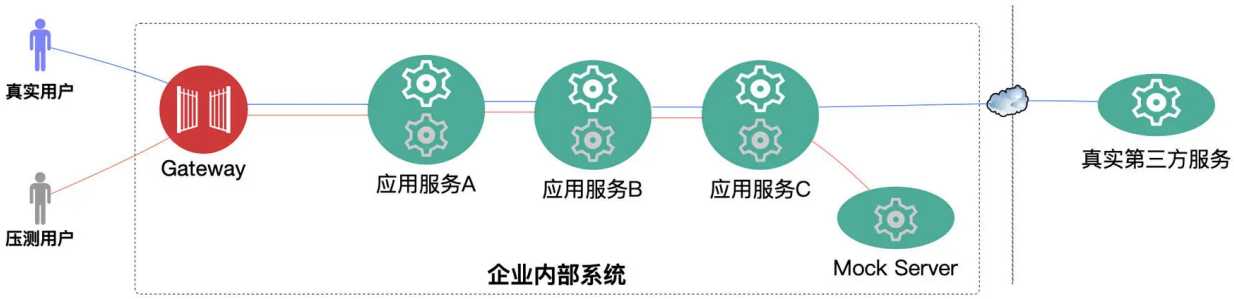
```
09 十一月 2021 13:29:07 [nioEventLoopGroup-3-2] INFO  Request received:

GET /7d HTTP/1.1
Host: 127.0.0.1:12306
User-Agent: PostmanRuntime/7.4.0
Accept: */*
Accept-Encoding: gzip, deflate
Cache-Control: no-cache
Content-Type: multipart/form-data; boundary=--------------------------46577703
Postman-Token: 791fa21c-386f-4389-aaa9-ba06d9e53aff
X-Lantern-Version: 5.1.0
Content-Length: 0

09 十一月 2021 13:29:07 [nioEventLoopGroup-3-2] INFO  Response return:

HTTP/1.1 200
```

刚才，我们对常见的接口类型进行了 Mock 自定义，你可以根据自己的接口类型进行相应的定义。

不过讲到这里，你可能还会有一个问题，这样的 Mock Server 可以支持大容量的压力吗？其实是完全可以的，你只要在前面加上一个 Nginx，后面的节点数是可以随便添加的，能一直加到满足你的需求为止。

系统改造

好了，知道了怎么搭建 Mock Server 之后，现在我们要来解决另一个关键问题，那就是如何基于压测标记实现流量的区分。还记得我们在这节课开头的这张图吧。我们希望，在同一个应用服务中实现正式流量和压测流量的请求（如果你是用 Service Mesh 做的不同版本的发布，不在这个逻辑之内）。

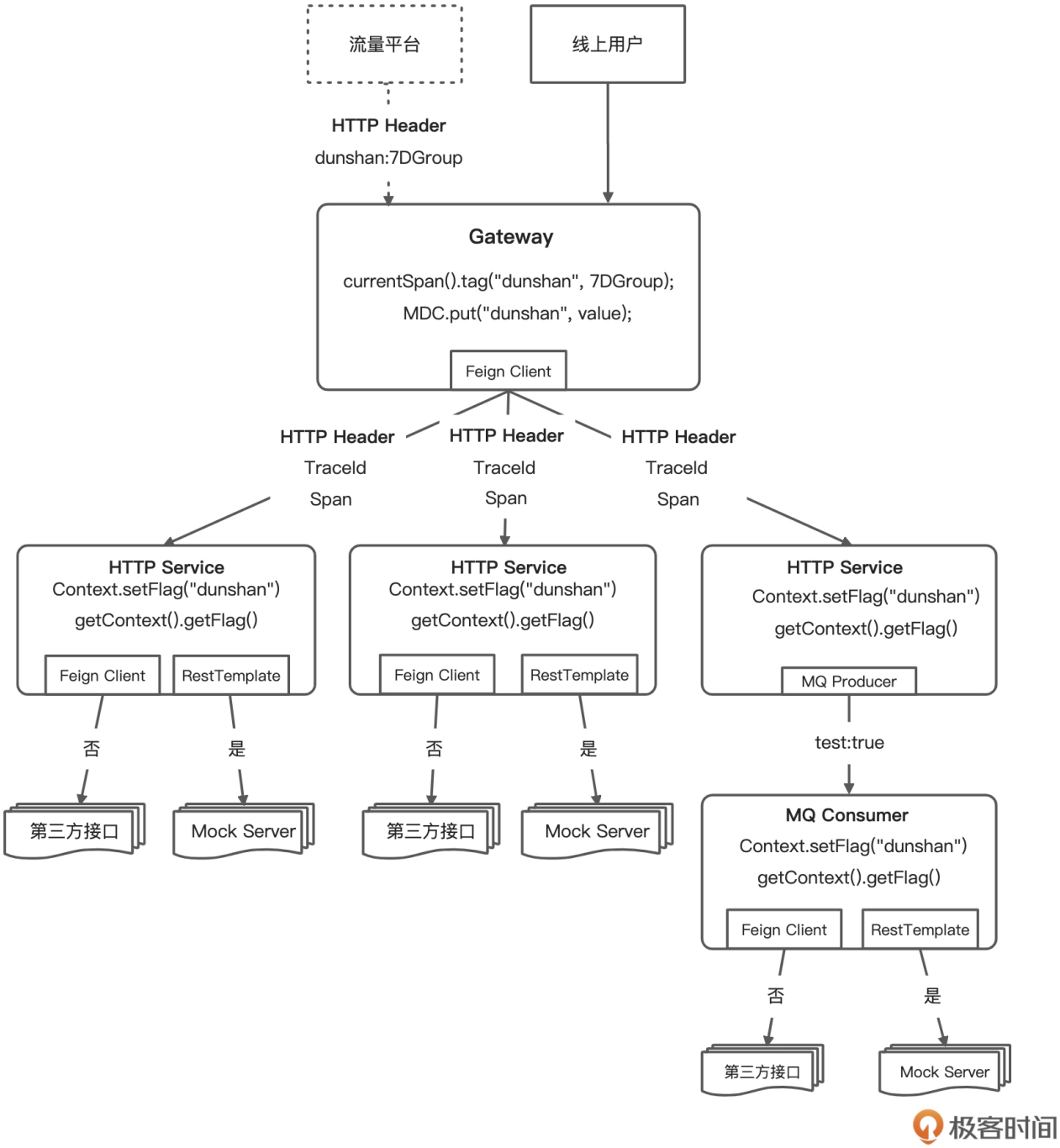


注意哦，重点来了！ 请看上面的应用服务 C，现在网上我们能看到的几乎所有 Mock 相关的文章，都是直接把这个应用服务 C 的后端调用接到了 Mock Server 上。其实这是有问题的，也不符合全链路压测的逻辑。

全链路压测的前提是不影响真实的生产服务的同时，又可以在同一应用服务中实现将压测流量转发到 Mock Server 上去。所以这时，就必须在应用服务 C 上做逻辑判断。也就是让正式的流量走真实的第三方服务，让压测流量走到 Mock Server 中去。

这就是我们需要改造的部分了。因为改造比较简单，这次我们不做 demo 了，直接在真实系统中改造。

先来看下我们的整体逻辑设计图：



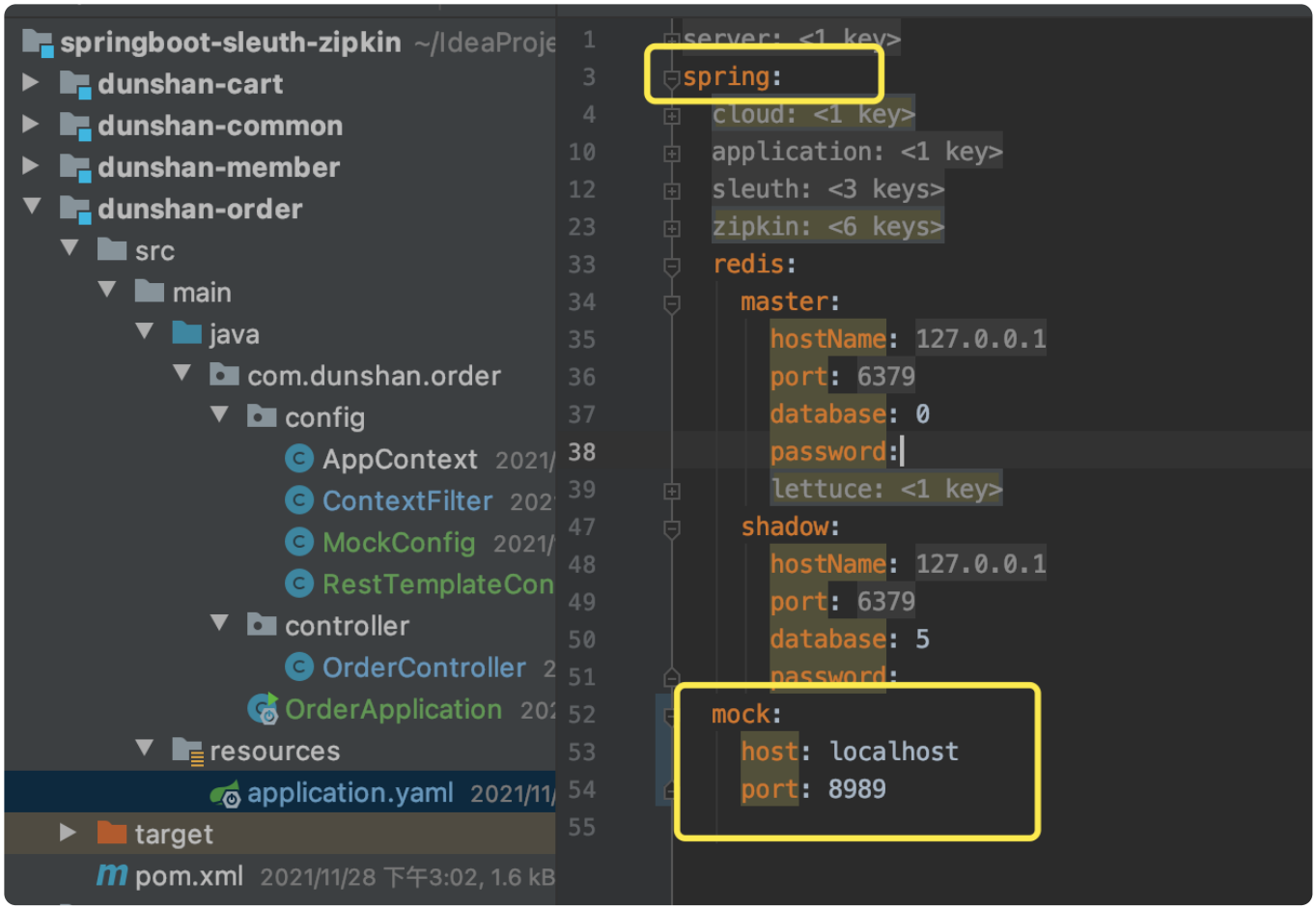
在这张图里，HTTP Header 透传到 Service 里后，业务方通过数据上下文获取到压测标记，这时我们要进行判断：

如果是正式流量，就走 Feign Client 调用真实的第三方接口；

如果是压测流量，就走 RestTemplate 调用 Mock Server。

我们用 order 服务中典型的需要做第三方 Mock 的 pay 接口来详细演示一下实现步骤。

第一步，在应用服务项目的配置文件 application.yaml 中添加 Mock 相关的配置。



第二步，编写 Mock 配置读取类。

复制代码

```
@Log4j2
@Component
public class MockConfig {

    @Value("${spring.mock.host}")
    private String host;

    @Value("${spring.mock.port}")
    private String port;

    public String getHost() {
        return host;
    }

    public String getPort() {
        return port;
    }

}
```

第三步，编写 RestTemplate 配置类，用于后面的 HTTP 调用。

复制代码

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.client.ClientHttpRequestFactory;
import org.springframework.http.client.SimpleClientHttpRequestFactory;
import org.springframework.web.client.RestTemplate;

/**
 * @description:  RestTemplate 配置类
 * @author: dunshan
 * @create: 2021-11-28 16:23
 **/

@Configuration
public class RestTemplateConfig {

    @Bean
    public RestTemplate restTemplate(ClientHttpRequestFactory factory) {
        return new RestTemplate(factory);
    }

    @Bean
    public ClientHttpRequestFactory simpleClientHttpRequestFactory() {
        SimpleClientHttpRequestFactory factory = new SimpleClientHttpRequestFa
        factory.setReadTimeout(5000);
        factory.setConnectTimeout(15000);
        return factory;
    }

}
```

第四步，在调用 pay 接口时实现 Mock 判断。

复制代码

```

private SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

@Autowired
RestTemplate restTemplate;
@Autowired
MockConfig mockConfig;

@GetMapping("/pay")
public Object payOrder() {

```

```
    HashMap<String, Object> map = new HashMap<>();
    String name = "";

    //获取流量标识
    String flag = AppContext.getContext().getFlag();

    if (StringUtils.isNoneEmpty(flag)) {
         //请求 mock 服务
        String url = String.format("http://%s:%s/7d", mockConfig.getHost(), mo
        ResponseEntity<String> response = restTemplate.getForEntity(url, Strin
        String body = response.getBody();
        map.put("ok", JSON.parse(body));
        map.put("time", sdf.format(new Date()));
        return map;
    }

    //走正常服务
    map.put("ok", "正式数据");
    return map;

}
```

第五步，Moco 服务响应 JSON 配置。

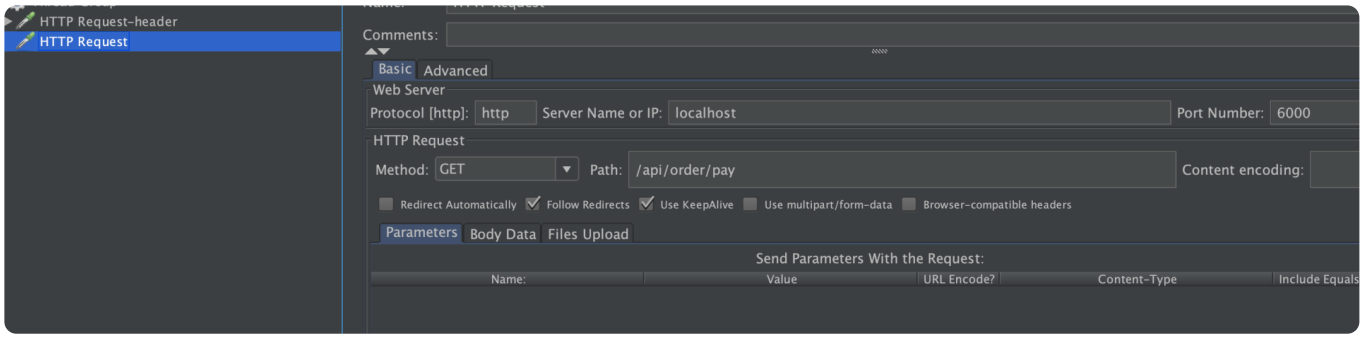
复制代码

```
[
  {
    "description": "这是一个get请求",
    "request": {
      "uri": "/7d",
      "method": "get"
    },
    "response": {
      "json":{
        "name":"Mock Success",
        "code":"100"
      }
    }
  }
]

```

第六步，验证 Mock 服务。

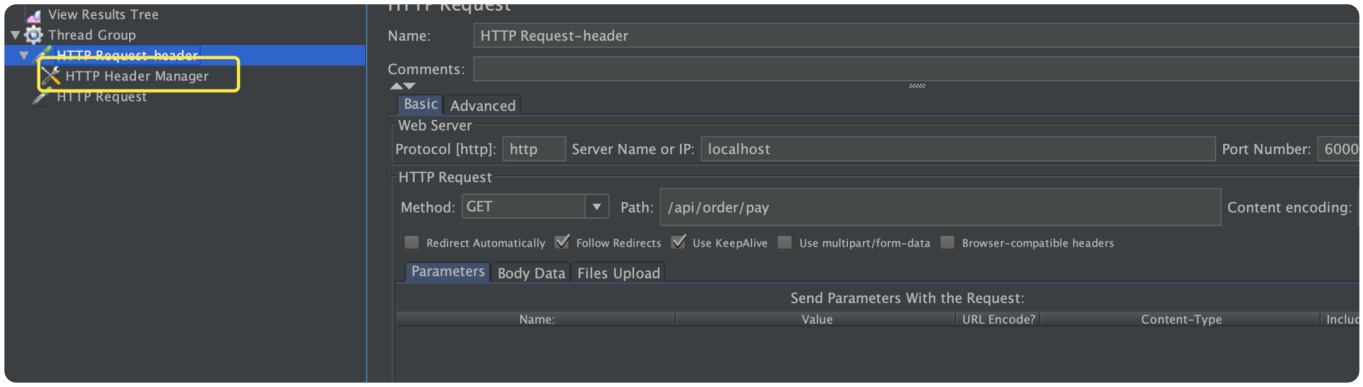
我们启动 JMeter，模拟发送正常流量和压测流量。

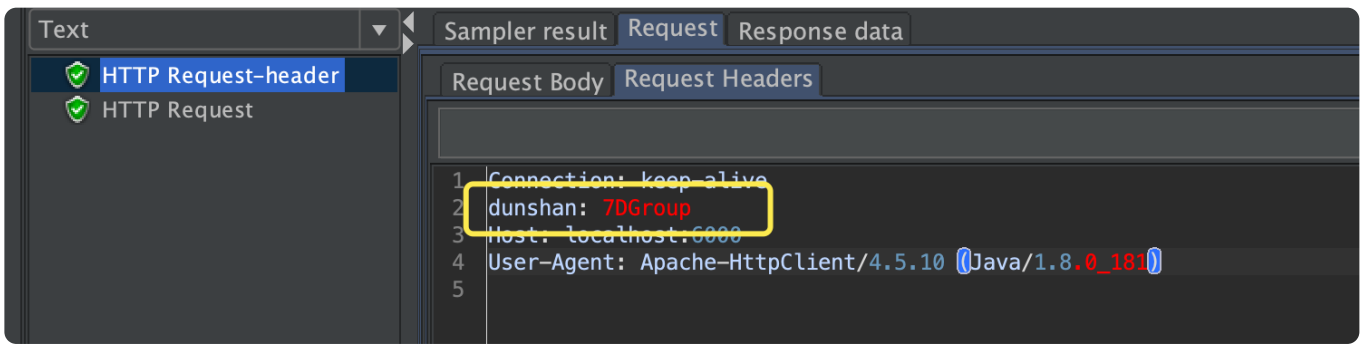
正常流量（不带压测标记）



返回结果如下：

压测流量（带压测标记）





返回结果如下：

从上面的结果可以看到，压测流量请求会走到 Mock Server 上；而正常的流量请求会走到真正的第三方服务上。

到这里，全链路压测的 Mock 改造就成功了。

总结

好，这节课就讲到这里。

刚才，我们介绍了常见的前端和后端 Mock Server，并做了详细的演示。我们还演示了系统改造的部分。主要的逻辑，就是业务方从数据上下文中获取压测标记，然后通过判断让它走不同的调用方向。**这里的关键是让压测流量和正常流量在同一个服务中去实现**。这样在全链路线上压测时，才能真正地把所有生产上用到的服务和方法都覆盖到了。

现在市场上的 Mock Server 已经有很多了。至于用哪一种工具其实并不重要，只要能满足需求就可以。如果没有 Mock Server，我们还可以自己写一个服务。

Mock 是一种比较简单易懂的逻辑，但它却能解决全链路压测中对第三方强依赖的问题，这可解决了一个大问题，希望你能够用好它。

思考题

学完这节课，我想请你思考两个问题：

1. 你还知道哪些可以作为 Mock Server 的应用？各有什么优缺点？

2. 你觉得 Mock Server 的配置中，还有哪些难点？

欢迎你在留言区与我交流讨论。我们下节课见！

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 1 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 19 | 日志隔离：如何落地日志隔离？

下一篇 21 | 压测平台：高效搭建 GoReplay 压测平台

精选留言

写留言

> 由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。